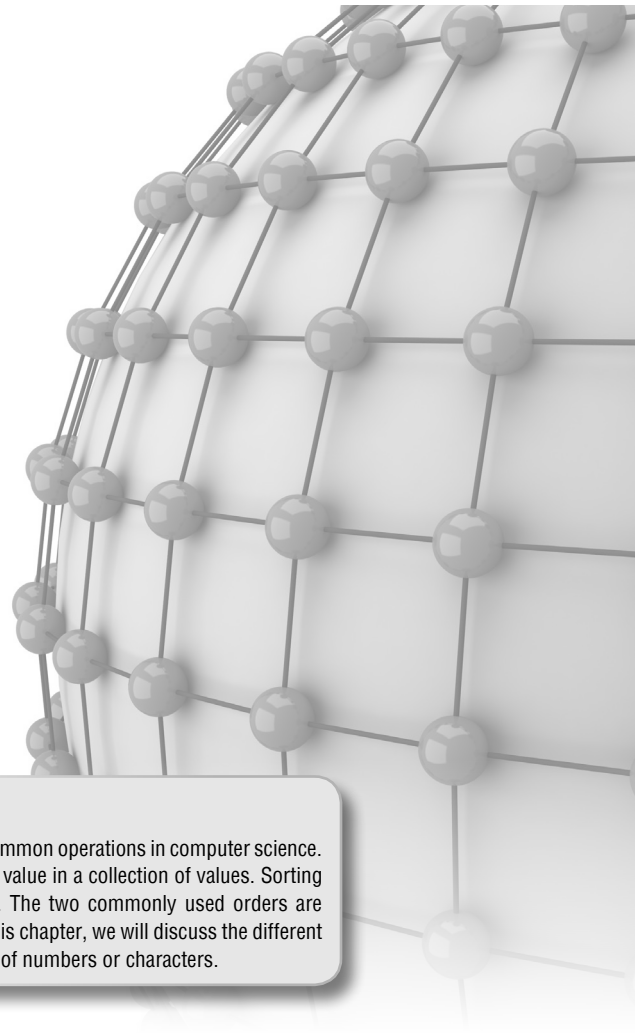
CHAPTER 14

Searching and Sorting



LEARNING OBJECTIVE

Searching and sorting are two of the most common operations in computer science. Searching refers to finding the position of a value in a collection of values. Sorting refers to arranging data in a certain order. The two commonly used orders are numerical order and alphabetical order. In this chapter, we will discuss the different techniques of searching and sorting arrays of numbers or characters.

14.1 INTRODUCTION TO SEARCHING

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
There are two popular methods for searching the array elements: *linear search* and *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity. We will discuss all these methods in detail in this section.

14.2 LINEAR SEARCH

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array `A[]` is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = –1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Figure 14.1 shows the algorithm for linear search.

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

Programming Example

1. Write a program to search an element in an array using the linear search technique.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, found = 0, pos = -1;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number that has to be searched : ");
        scanf("%d", &num);
        for(i=0;i<n;i++)
        {
                if(arr[i] == num)
                {
                        found =1;
                        pos=i;
                        printf("\n %d is found in the array at position= %d", num,i+1);
                        /* +1 added in line 23 so that it would display the number in
              the first place in the array as in position 1 instead of 0 */
                        break;
                }
        }
        if (found == 0)
```

```
            printf("\n %d does not exist in the array", num);
            return 0;
    }
```

14.3 BINARY SEARCH

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array. Consider an array `A[]` that is declared and initialized as

```
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the value to be searched is `VAL = 9`. The algorithm will proceed in the following manner.

```
BEG = 0, END = 10, MID = (0 + 10)/2 = 5
```

Now, `VAL = 9` and `A[MID] = A[5] = 5`

`A[5]` is less than `VAL`, therefore, we now search for the value in the second half of the array. So, we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8`

```
VAL = 9 and A[MID] = A[8] = 8
```

`A[8]` is less than `VAL`, therefore, we now search for the value in the second half of the segment. So, again we change the values of `BEG` and `MID`.

Now, `BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9`

Now, `VAL = 9` and `A[MID] = 9`.

In this algorithm, we see that `BEG` and `END` are the beginning and ending positions of the segment that we are looking to search for the element. `MID` is calculated as `(BEG + END)/2`. Initially, `BEG = lower_bound` and `END = upper_bound`. The algorithm will terminate when `A[MID] = VAL`. When the algorithm ends, we will set `POS = MID`. `POS` is the position at which the value is present in the array.

However, if `VAL` is not equal to `A[MID]`, then the values of `BEG`, `END`, and `MID` will be changed depending on whether `VAL` is smaller or greater than `A[MID]`.

(a) If `VAL < A[MID]`, then `VAL` will be present in the left segment of the array. So, the value of `END` will be changed as `END = MID – 1`.

(b) If `VAL > A[MID]`, then `VAL` will be present in the right segment of the array. So, the value of `BEG` will be changed as `BEG = MID + 1`.

Finally, if `VAL` is not present in the array, then eventually, `END` will be less than `BEG`. When this happens, the algorithm will terminate and the search will be unsuccessful.

Figure 14.2 shows the algorithm for binary search.

In Step 1, we initialize the value of variables, `BEG`, `END`, and `POS`. In Step 2, a `while` loop is executed until `BEG` is less than or equal to `END`. In Step 3, the value of `MID` is calculated. In Step 4, we check if the array value at `MID` is equal to `VAL` (item to be searched in the array). If a match is

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.2 Algorithm for binary search

found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered. In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Complexity of Binary Search Algorithm

The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$2^{f(n)} > n$ or $f(n) = \log_2 n$

Programming Example

2. Write a program to search an element in an array using binary search.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10 // Added to make changing size of array easier
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, beg, end, mid, found=0;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);  // Added to sort the array
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        printf("\n\n Enter the number that has to be searched: ");
        scanf("%d", &num);
        beg = 0, end = n-1;
        while(beg<=end)
        {
                mid = (beg + end)/2;
                if (arr[mid] == num)
                {
                        printf("\n %d is present in the array at position %d", num, mid+1);
                        found =1;
                        break;
                }
```

```
                    else if (arr[mid]>num)
                    end = mid-1;
                    else
                    beg = mid+1;
            }
            if (beg > end && found == 0)
            printf("\n %d does not exist in the array", num);
            return 0;
    }

    int smallest(int arr[], int k, int n)
    {
            int pos = k, small=arr[k], i;
            for(i=k+1;i<n;i++)
            {
                    if(arr[i]< small)
                    {
                            small = arr[i];
                            pos = i;
                    }
            }
            return pos;
    }
    void selection_sort(int arr[],int n)
    {
            int k, pos, temp;
            for(k=0;k<n;k++)
            {
                    pos = smallest(arr, k, n);
                    temp = arr[k];
                    arr[k] = arr[pos];
                    arr[pos] = temp;
            }
    }
```

14.4 INTERPOLATION SEARCH

Interpolation search, also known as extrapolation search, is a searching technique that finds a specified value in a sorted array. The concept of interpolation search is similar to how we search for names in a telephone book or for keys by which a book's entries are ordered. For example, when looking for a name "Bharat" in a telephone directory, we know that it will be near the extreme left, so applying a binary search technique by dividing the list in two halves each time is not a good idea. We must start scanning the extreme left in the first pass itself.

In each step of interpolation search, the remaining search space for the value to be found is calculated. The calculation is done based on the values at the bounds of the search space and the value to be searched. The value found at this estimated position is then compared with the value being searched for. If the two values are equal, then the search is complete.

```
INTERPOLATION_SEARCH (A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET LOW = lower_bound,
        HIGH = upper_bound, POS = -1
Step 2:    Repeat Steps 3 to 4 while LOW <= HIGH
Step 3:         SET MID = LOW + (HIGH - LOW) ×
                ((VAL - A[LOW]) / (A[HIGH] - A[LOW]))
Step 4:         IF VAL = A[MID]
                  POS = MID
                  PRINT POS
                  Go to Step 6
                 ELSE IF VAL < A[MID]
                   SET HIGH = MID - 1
                 ELSE
                   SET LOW = MID + 1
                [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.3 Algorithm for interpolation search

However, in case the values are not equal then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position. Thus, we see that interpolation search is similar to the binary search technique. However, the important difference between the two techniques is that binary search always selects the middle value of the remaining search space. It discards half of the values based on the comparison between the value found at the estimated position and the value to be searched. But in interpolation search, interpolation is used to find an item near the one being searched for, and then linear search is used to find the exact item.

The algorithm for interpolation search is given in Fig. 14.3.

Figure 14.4 helps us to visualize how the search space is divided in case of binary search and interpolation search.

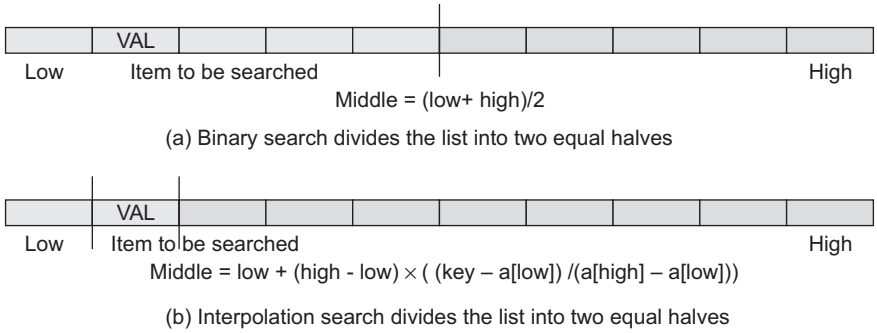


Figure 14.4 Difference between binary search and interpolation search

Complexity of Interpolation Search Algorithm

When `n` elements of a list to be sorted are uniformly distributed (average case), interpolation search makes about `log(log n)` comparisons. However, in the worst case, that is when the elements increase exponentially, the algorithm can make up to `O(n)` comparisons.

Example 14.1 Given a list of numbers `a[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}`. Search for value 19 using interpolation search technique.

Solution

```
Low = 0, High = 10, VAL = 19, a[Low] = 1, a[High] = 21
Middle = Low + (High – Low)×((VAL – a[Low]) /(a[High] – a[Low] ))
       = 0 +(10 – 0) × ((19 – 1) / (21 – 1) )
       = 0 + 10 × 0.9 = 9
a[middle] = a[9] = 19 which is equal to value to be searched.
```

Programming Example

3. Write a program to search an element in an array using interpolation search.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int interpolation_search(int a[], int low, int high, int val)
{
        int mid;
        while(low <= high)
        {
                mid = low + (high – low)*((val – a[low]) / (a[high] – a[low]));
                if(val == a[mid])
                      return mid;
                if(val < a[mid])
```

```
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return -1;
}
int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = interpolation_search(arr, 0, n-1, val);
        if(pos == -1)
                printf("\n %d is not found in the array", val);
        else
                printf("\n %d is found at position %d", val, pos);
        getche();
        return 0;
}
```

14.5 JUMP SEARCH

When we have an already sorted list, then the other efficient algorithm to search for a value is jump search or block search. In jump search, it is not necessary to scan all the elements in the list to find the desired value. We just check an element and if it is less than the desired value, then some of the elements following it are skipped by jumping ahead. After moving a little forward again, the element is checked. If the checked element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element. However, if the checked element is less than the value being searched for, then we again make a small jump and repeat the process.

Once the boundary of the value is determined, a linear search is done to find the value and its position in the array. For example, consider an array `a[] = {1,2,3,4,5,6,7,8,9}`. The length of the array is 9. If we have to find value 8 then following steps are performed using the jump search technique.

Step 1: First three elements are checked. Since 3 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 2: Next three elements are checked. Since 6 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 3: Next three elements are checked. Since 9 is greater than 8, the desired value lies within the current boundary

1	2	3	4	5	6	7	8	9

Step 4: A linear search is now done to find the value in the array.

The algorithm for jump search is given in Fig. 14.5.

```
JUMP_SEARCH (A, lower_bound, upper_bound, VAL, N)

Step 1: [INITIALIZE] SET STEP = sqrt(N), I = 0, LOW = lower_bound, HIGH = upper_bound, POS = -1
Step 2: Repeat Step 3 while I < STEP
Step 3:        IF VAL < A[STEP]
                 SET HIGH = STEP - 1
               ELSE
                 SET LOW = STEP + 1
               [END OF IF]
               SET I = I + 1
         [END OF LOOP]
Step 4: SET I = LOW
Step 5: Repeat Step 6 while I <= HIGH
Step 6:          IF A[I] = Val
                        POS =  I
                        PRINT POS
                        Go to Step 8
                 [END OF IF]
                 SET I = I + 1
         [END OF LOOP]
Step 7:  IF POS = -1
               PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
         [END OF IF]
Step 8: EXIT
```

Figure 14.5 Algorithm for jump search

Advantage of Jump Search over Linear Search

Suppose we have a sorted list of 1000 elements where the elements have values `0, 1, 2, 3, 4, …, 999`, then sequential search will find the value 674 in exactly 674 iterations. But with jump search, the same value can be found in 44 iterations. Hence, jump search performs far better than a linear search on a sorted list of elements.

Advantage of Jump Search over Binary Search

No doubt, binary search is very easy to implement and has a complexity of `O(log n)`, but in case of a list having very large number of elements, jumping to the middle of the list to make comparisons is not a good idea because if the value being searched is at the beginning of the list then one (or even more) large step(s) in the backward direction would have to be taken. In such cases, jump search performs better as we have to move little backward that too only once. Hence, when jumping back is slower than jumping forward, the jump search algorithm always performs better.

How to Choose the Step Length?

For the jump search algorithm to work efficiently, we must define a fixed size for the step. If the step size is 1, then algorithm is same as linear search. Now, in order to find an appropriate step size, we must first try to figure out the relation between the size of the list (`n`) and the size of the step (`k`). Usually, `k` is calculated as $\sqrt{n}$.

Further Optimization of Jump Search

Till now, we were dealing with lists having small number of elements. But in real-world applications, lists can be very large. In such large lists searching the value from the beginning of the list may not be a good idea. A better option is to start the search from the `k`-th element as shown in the figure below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	….

Searching can start from somewhere middle in the list rather than from the beginning to optimize performance.

We can also improve the performance of jump search algorithm by repeatedly applying jump search. For example, if the size of the list is 10,00,000 (n). The jump interval would then be √n = √1000000 =1000. Now, even the identified interval has 1000 elements and is again a large list. So, jump search can be applied again with a new step size of √1000 ≈ 31. Thus, every time we have a desired interval with a large number of values, the jump search algorithm can be applied again but with a smaller step. However, in this case, the complexity of the algorithm will no longer be O(√n) but will approach a logarithmic value.

Complexity of Jump Search Algorithm

Jump search works by jumping through the array with a step size (optimally chosen to be √n) to find the interval of the value. Once this interval is identified, the value is searched using the linear search technique. Therefore, the complexity of the jump search algorithm can be given as **O(√n)**.

Programming Example

4. Write a program to search an element in an array using jump search.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#define MAX 20
int jump_search(int a[], int low, int high, int val, int n)
{
        int step, i;
        step = sqrt(n);
        for(i=0;i<step;i++)
        {
                if(val < a[step])
                        high = step – 1;
                else
                        low = step + 1;
        }
        for(i=low;i<=high;i++)
        {
             if(a[i]==val)
                  return i;
        }
        return –1;
}

int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = jump_search(arr, 0, n–1, val, n);
```

```
            if(pos == -1)
                    printf("\n %d is not found in the array", val);
            else
                    printf("\n %d is found at position %d", val, pos);
            getche();
            return 0;
    }
```

Fibonacci Search

We are all well aware of the Fibonacci series in which the first two terms are 0 and 1 and then each successive term is the sum of previous two terms. In the Fibonacci series given below, each number is called a Fibonacci number.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The same series and concept can be used to search for a given value in a list of numbers. Such a search algorithm which is based on Fibonacci numbers is called Fibonacci search and was developed by Kiefer in 1953. The search follows a divide-and-conquer technique and narrows down possible locations with the help of Fibonacci numbers.

Fibonacci search is similar to binary search. It also works on a sorted list and has a run time complexity of `O(log n)`. However, unlike the binary search algorithm, Fibonacci search does not divide the list into two equal halves rather it subtracts a Fibonacci number from the index to reduce the size of the list to be searched. So, the key advantage of Fibonacci search over binary search is that comparison dispersion is low.

14.6 INTRODUCTION TO SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if `A` is an array, then the elements of `A` are arranged in a sorted order (ascending order) in such a way that `A[0] < A[1] < A[2] < ..... < A[N]`.

For example, if we have an array that is declared and initialized as

```
int A[] = {21, 34, 11, 9, 1, 0, 22};
```

Then the sorted array (ascending order) can be given as:

```
A[] = {0, 1, 9, 11, 21, 22, 34;
```

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types of sorting:

- ***Internal sorting*** which deals with sorting the data stored in the computer's memory
- ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

14.6.1 Sorting on Multiple Keys

Many a times, when performing real-world applications, it is desired to sort arrays of records using multiple keys. This situation usually occurs when a single key is not sufficient to uniquely identify a record. For example, in a big organization we may want to sort a list of employees on the basis of their departments first and then according to their names in alphabetical order.

Other examples of sorting on multiple keys can be

- Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order.

- In a library, the information about books can be sorted alphabetically based on titles and then by authors' names.
- Customers' addresses can be sorted based on the name of the city and then the street.

Note Data records can be sorted based on a property. Such a component or property is called a **sort key**. A sort key can be defined using two or more sort keys. In such a case, the first key is called the **primary sort key**, the second is known as the **secondary sort key**, etc.

Consider the data records given below:

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

Now if we take department as the primary key and name as the secondary key, then the sorted order of records can be given as:

Name	Department	Salary	Phone Number
Divya	Computer Science	750000	9350123455
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Janak	Telecommunications	1000000	9812345678

Observe that the records are sorted based on department. However, within each department the records are sorted alphabetically based on the names of the employees.

14.6.2 Practical Considerations for Internal Sorting

As mentioned above, records can be sorted either in ascending or descending order based on a field often called as the sort key. The list of records can be either stored in a contiguous and randomly accessible data structure (array) or may be stored in a dispersed and only sequentially accessible data structure like a linked list. But irrespective of the underlying data structure used to store the records, the logic to sort the records will be same and only the implementation details will differ.

When analysing the performance of different sorting algorithms, the practical considerations would be the following:

- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst case performance
- Average case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

14.7 BUBBLE SORT

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements

in ascending order). In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Note If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Technique

The basic methodology of the working of bubble sort is given as follows:

(a) In Pass 1, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-2]` is compared with `A[N-1]`. Pass 1 involves `n-1` comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-3]` is compared with `A[N-2]`. Pass 2 involves `n-2` comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-4]` is compared with `A[N-3]`. Pass 3 involves `n-3` comparisons and places the third biggest element at the third highest index of the array.

(d) In Pass `n-1`, `A[0]` and `A[1]` are compared so that `A[0]<A[1]`. After this step, all the elements of the array are arranged in ascending order.

Example 14.2 To discuss bubble sort in detail, let us consider an array `A[]` that has the following elements:

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
 (a) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (b) Compare 52 and 29. Since 52 > 29, swapping is done.
     30, 29, 52, 87, 63, 27, 19, 54
 (c) Compare 52 and 87. Since 52 < 87, no swapping is done.
 (d) Compare 87 and 63. Since 87 > 63, swapping is done.
     30, 29, 52, 63, 87, 27, 19, 54
 (e) Compare 87 and 27. Since 87 > 27, swapping is done.
     30, 29, 52, 63, 27, 87, 19, 54
 (f) Compare 87 and 19. Since 87 > 19, swapping is done.
     30, 29, 52, 63, 27, 19, 87, 54
 (g) Compare 87 and 54. Since 87 > 54, swapping is done.
     30, 29, 52, 63, 27, 19, 54, 87
```

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

```
Pass 2:
 (a) Compare 30 and 29. Since 30 > 29, swapping is done.
     29, 30, 52, 63, 27, 19, 54, 87
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 63. Since 52 < 63, no swapping is done.
 (d) Compare 63 and 27. Since 63 > 27, swapping is done.
     29, 30, 52, 27, 63, 19, 54, 87
 (e) Compare 63 and 19. Since 63 > 19, swapping is done.
```

```
    29, 30, 52, 27, 19, 63, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
    29, 30, 52, 27, 19, 54, 63, 87
```

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

```
Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
    29, 30, 27, 52, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
    29, 30, 27, 19, 52, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.
```

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

```
Pass 4:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
    29, 27, 30, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
    29, 27, 19, 30, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.
```

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

```
Pass 5:
(a) Compare 29 and 27. Since 29 > 27, swapping is done.
    27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
    27, 19, 29, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.
```

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

```
Pass 6:
(a) Compare 27 and 19. Since 27 > 19, swapping is done.
    19, 27, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.
```

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

```
Pass 7:
(a) Compare 19 and 27. Since 19 < 27, no swapping is done.
```

Observe that the entire list is sorted now.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Figure 14.6 Algorithm for bubble sort

Figure 14.6 shows the algorithm for bubble sort. In this algorithm, the outer loop is for the total number of passes which is `N-1`. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed `N-I` times, where `N` is the number of elements in the array and `I` is the count of the pass.

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are N–1 passes in total. In the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$f(n) = (n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1$
$f(n) = n\ (n - 1)/2$
$f(n) = n^2/2 + O(n) = O(n^2)$

Therefore, the complexity of bubble sort algorithm is $O(n^2)$. It means the time required to execute bubble sort is proportional to n^2, where n is the total number of elements in the array.

PROGRAMMING EXAMPLE

5. Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, temp, j, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr [i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                        }
                }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
                printf("%d\t", arr[i]);
        getch();
        return 0;
}
Output
Enter the number of elements in the array : 10
Enter the elements : 8    9    6    7    5    4    2    3    1    10
The array sorted in ascending order is :
1    2    3    4    5    6    7    8    9    10
```

Bubble Sort Optimization

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n–1 passes. We may even have an array that will be sorted in 2 or 3

passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to TRUE before each pass and is made FALSE when a swapping is performed. The code for the optimized bubble sort can be given as:

```
void bubble_sort(int *arr, int n)
{
     int i, j, temp, flag = 0;
     for(i=0; i<n; i++)
    {
       for(j=0; j<n-i-1; j++)
       {
          if(arr[j]>arr[j+1])
          {
             flag = 1;
             temp = arr[j+1];
             arr[j+1] = arr[j];
             arr[j] = temp;
          }
       }
      if(flag == 0)   // array is sorted
         return;
   }
}
```

Complexity of Optimized Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take $O(n)$ time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In average case also, the performance will see an improvement. Compare it with the complexity of original bubble sort algorithm which takes $O(n^2)$ in all the cases.

14.8 INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

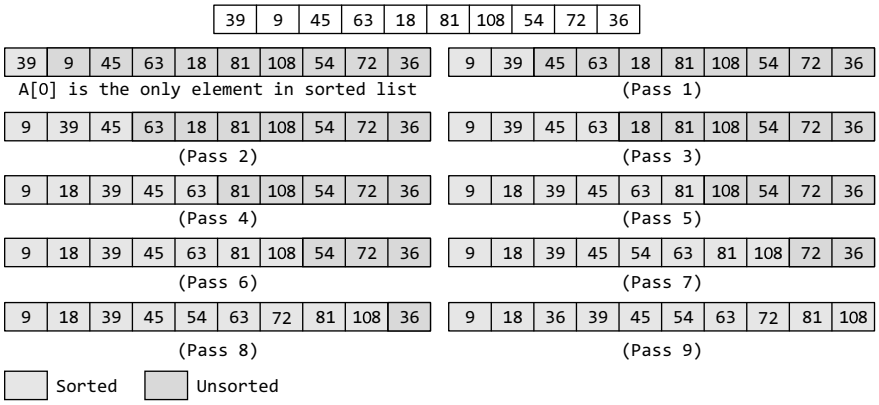
Technique

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming $LB = 0$) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if $LB = 0$).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example 14.3 Consider an array of integers given below. We will sort the values in the array using insertion sort.

Solution



Initially, `A[0]` is the only element in the sorted set. In Pass 1, `A[1]` will be placed either before or after `A[0]`, so that the array `A` is sorted. In Pass 2, `A[2]` will be placed either before `A[0]`, in between `A[0]` and `A[1]`, or after `A[1]`. In Pass 3, `A[3]` will be placed in its proper place. In Pass `N-1`, `A[N-1]` will be placed in its proper place to keep the array sorted.

To insert an element `A[K]` in a sorted list `A[0]`, `A[1]`, ..., `A[K-1]`, we need to compare `A[K]` with `A[K-1]`, then with `A[K-2]`, `A[K-3]`, and so on until we meet an element `A[J]` such that `A[J] <= A[K]`. In order to insert `A[K]` in its correct position, we need to move elements `A[K-1]`, `A[K-2]`, ..., `A[J]` by one position and then `A[K]` is inserted at the `(J+1)`th location. The algorithm for insertion sort is given in Fig. 14.7.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:     SET TEMP = ARR[K]
Step 3:     SET J = K - 1
Step 4:     Repeat while TEMP <= ARR[J]
                SET ARR[J + 1] = ARR[J]
                SET J = J - 1
            [END OF INNER LOOP]
Step 5:     SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Figure 14.7 Algorithm for insertion sort

In the algorithm, Step 1 executes a `for` loop which will be repeated for each element in the array. In Step 2, we store the value of the `K`th element in `TEMP`. In Step 3, we set the `J`th index in the array. In Step 4, a `for` loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the `(J+1)th` location.

Complexity of Insertion Sort

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., `O(n)`). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., $O(n^2)$).

Even in the average case, the insertion sort algorithm will have to make at least `(K-1)/2` comparisons. Thus, the average case also has a quadratic running time.

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only $O(1)$ of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

PROGRAMMING EXAMPLE

6. Write a program to sort an array using insertion sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        insertion_sort(arr, n);
        printf("\n The sorted array is:  \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void insertion_sort(int arr[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
                temp = arr[i];
                j = i-1;
                while((temp < arr[j]) && (j>=0))
                {
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = temp;
        }
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 500 1 50 23 76
The sorted array is :
1    23    20    76    500    6    7    8    9    10
```

14.9 SELECTION SORT

Selection sort is a sorting algorithm that has a quadratic running time complexity of $O(n^2)$, thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over

more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique

Consider an array ARR with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
- In Pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1]. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], . . . , ARR[N-1] is sorted.

Example 14.4 Sort the array given below using selection sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

The algorithm for selection sort is shown in Fig. 14.8. In the algorithm, during the K^{th} pass, we need to find the position POS of the smallest elements from ARR[K], ARR[K+1], . . . , ARR[N]. To find the smallest element, we use a variable SMALL to hold the smallest value in the sub-array ranging from ARR[K] to ARR[N]. Then, swap ARR[K] with ARR[POS]. This procedure is repeated until all the elements in the array are sorted.

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Figure 14.8 Algorithm for selection sort

Complexity of Selection Sort

Selection sort is a sorting algorithm that is independent of the original order of elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements;

thus, n–1 comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on. Therefore,

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$
$$= n(n - 1) / 2 = O(n^2) \text{ comparisons}$$

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Programming Example

7. Write a program to sort an array using selection sort algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int smallest(int arr[], int k, int n);
void selection_sort(int arr[], int n);
void main(int argc, char *argv[]) {
        int arr[10], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
}
int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}
void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];
```

```
                    arr[k] = arr[pos];
                    arr[pos] = temp;
                }
    }
```

14.10 MERGE SORT

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the `n`-element array to be sorted into two sub-arrays of `n/2` elements. If `A` is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide `A` into two sub-arrays, A_1 and A_2, each containing about half of the elements of `A`.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size `n/2` to produce the sorted array of `n` elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

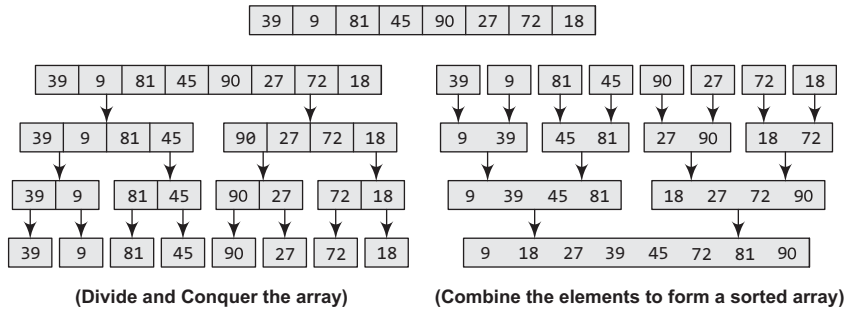
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

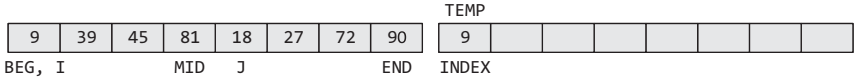
Example 14.5 Sort the array given below using merge sort.

Solution

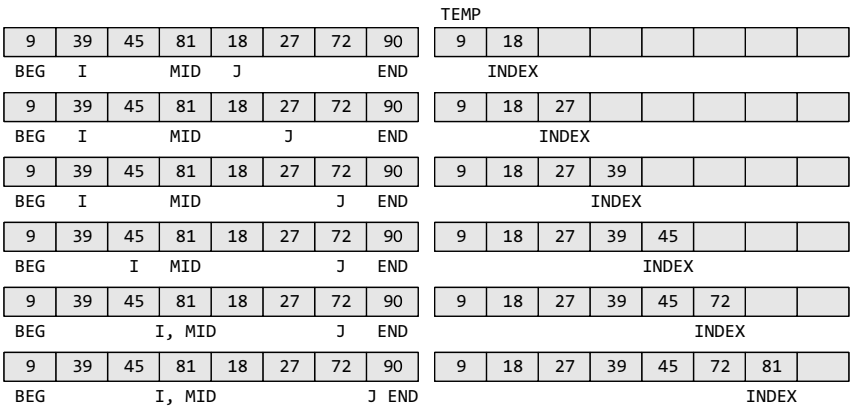


The merge sort algorithm (Fig. 14.9) uses a function `merge` which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

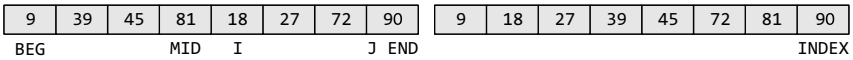
To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.



Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.



When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.




```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = INDEX + 1, SET I = I + 1
               [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

Figure 14.9 Algorithm for merge sort

The running time of merge sort in the average case and the worst case can be given as O(n log n). Although merge sort has an optimal time complexity, it needs an additional space of O(n) for the temporary array TEMP.

Programming Example

8. Write a program to implement merge sort.

```
#include <stdio.h>
#include <conio.h>
#define size 100

void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        merge_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void merge(int arr[], int beg, int mid, int end)
{
        int i=beg, j=mid+1, index=beg, temp[size], k;
        while((i<=mid) && (j<=end))
        {
                if(arr[i] < arr[j])
                {
                        temp[index] = arr[i];
                        i++;
                }
                else
                {
                        temp[index] = arr[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=end)
                {
```

```
                        temp[index] = arr[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = arr[i];
                        i++;
                        index++;
                }
        }
        for(k=beg;k<index;k++)
        arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
                mid = (beg+end)/2;
                merge_sort(arr, beg, mid);
                merge_sort(arr, mid+1, end);
                merge(arr, beg, mid, end);
        }
}
```

14.11 QUICK SORT

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes $O(n \log n)$ comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as $O(n^2)$. Basically, the quick sort algorithm is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

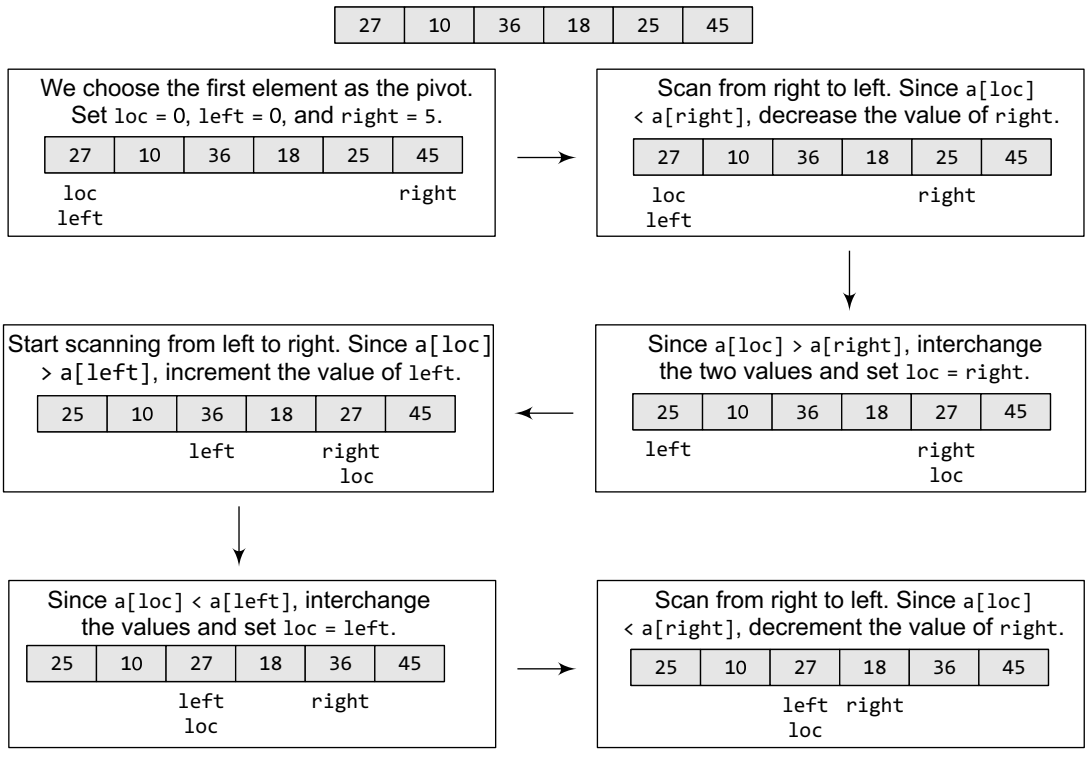
Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

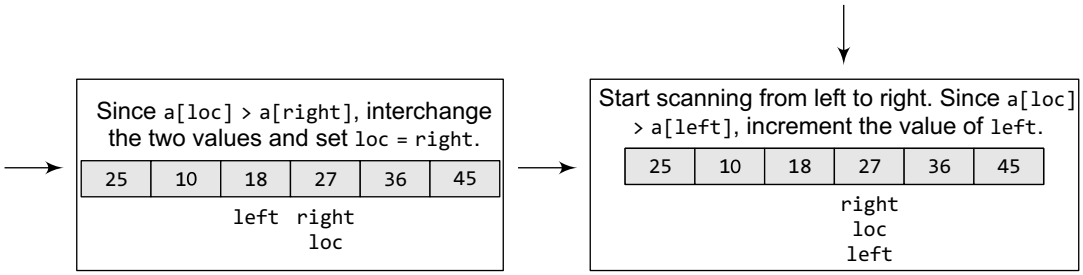
Technique

Quick sort works as follows:

1. Set the index of the first element in the array to `loc` and `left` variables. Also, set the index of the last element of the array to the `right` variable.
   That is, `loc = 0`, `left = 0`, and `right = n-1` (where `n` in the number of elements in the array)
2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable `loc`.
   That is, `a[loc]` should be less than `a[right]`.
   (a) If that is the case, then simply continue comparing until `right` becomes equal to `loc`. Once `right = loc`, it means the pivot has been placed in its correct position.
   (b) However, if at any point, we have `a[loc] > a[right]`, then interchange the two values and jump to Step 3.
   (c) Set `loc = right`
3. Start from the element pointed by `left` and scan the array from left to right, comparing each element on the way with the element pointed by `loc`.
   That is, `a[loc]` should be greater than `a[left]`.
   (a) If that is the case, then simply continue comparing until `left` becomes equal to `loc`. Once `left = loc`, it means the pivot has been placed in its correct position.
   (b) However, if at any point, we have `a[loc] < a[left]`, then interchange the two values and jump to Step 2.
   (c) Set `loc = left`.

Example 14.6 Sort the elements given in the following array using quick sort algorithm





Now `left = loc`, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

The quick sort algorithm (Fig. 14.10) makes use of a function `Partition` to divide the array into two sub-arrays.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
               SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
               SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
               SWAP ARR[LOC] with  ARR[RIGHT]
               SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
               Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
               [END OF LOOP]
Step 6:        IF LOC = LEFT
                       SET FLAG = 1
               ELSE IF ARR[LOC] < ARR[LEFT]
                       SWAP ARR[LOC] with  ARR[LEFT]
                       SET LOC = LEFT
               [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Figure 14.10 Algorithm for quick sort

Complexity of Quick Sort

In the average case, the running time of quick sort can be given as $O(n \log n)$. The partitioning of the array which simply loops over the elements of the array once uses $O(n)$ time.

In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only $\log n$ nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is $O(\log n)$. And because at each level, there can only be $O(n)$, the resultant time is given as $O(n \log n)$ time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot.

However, many implementations randomly choose the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of $O(n \log n)$.

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Programming Example

9. Write a program to implement quick sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                right--;
```

```
                    if(loc==right)
                    flag =1;
                    else if(a[loc]>a[right])
                    {
                            temp = a[loc];
                            a[loc] = a[right];
                            a[right] = temp;
                            loc = right;
                    }
                    if(flag!=1)
                    {
                            while((a[loc] >= a[left]) && (loc!=left))
                            left++;
                            if(loc==left)
                            flag =1;
                            else if(a[loc] <a[left])
                            {
                                    temp = a[loc];
                                    a[loc] = a[left];
                                    a[left] = temp;
                                    loc = left;
                            }
                    }
            }
            return loc;
    }
    void quick_sort(int a[], int beg, int end)
    {
            int loc;
            if(beg<end)
            {
                    loc = partition(a, beg, end);
                    quick_sort(a, beg, loc-1);
                    quick_sort(a, loc+1, end);
            }
    }
```

14.12 RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:           SET I = 0 and INITIALIZE buckets
Step 6:           Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT  = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                  [END OF LOOP]
Step 10:          Collect the numbers in the bucket
       [END OF LOOP]
Step 11: END
```

Figure 14.11 Algorithm for radix sort

Example 14.7 Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are `n` numbers that have to be sorted and `k` is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of `k` times. The inner loop is executed `n` times. Hence, the entire radix sort algorithm takes `O(kn)` time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in `O(n)` asymptotic time.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task.

Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.

Programming Example

10. Write a program to implement radix sort algorithm.

```
##include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
void main()
```

```
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        radix_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int largest(int arr[], int n)
{
        int large=arr[0], i;
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                large = arr[i];
        }
        return large;
}
void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                bucket_count[i]=0;
                for(i=0;i<n;i++)
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}

}
```

```
HEAPSORT(ARR, N)

Step 1: [Build Heap H]
      Repeat for I = 0 to N-1
            CALL Insert_Heap(ARR, N, ARR[I])
      [END OF LOOP]
Step 2: (Repeatedly delete the root element)
      Repeat while N>0
            CALL Delete_Heap(ARR, N, VAL)
            SET N = N + 1
      [END OF LOOP]
Step 3: END
```

Figure 14.12 Algorithm for heap sort

14.13 HEAP SORT

We have discussed binary heaps in Chapter 12. Therefore, we already know how to build a heap H from an array, how to insert a new element in an already existing heap, and how to delete an element from H. Now, using these basic concepts, we will discuss the application of heaps to write an efficient algorithm of heap sort (also known as tournament sort) that has a running time complexity of O(n log n).

Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

- In phase 1, build a heap H using the elements of ARR.
- In phase 2, repeatedly delete the root element of the heap formed in phase 1.

In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order. The algorithm of heap sort is shown in Fig. 14.12.

Complexity of Heap Sort

Heap sort uses two heap operations: *insertion* and *root deletion*. Each element extracted from the root is placed in the last empty location of the array.

In phase 1, when we build a heap, the number of comparisons to find the right location of the new element in H cannot exceed the depth of H. Since H is a complete tree, its depth cannot exceed m, where m is the number of elements in heap H.

Thus, the total number of comparisons g(n) to insert n elements of ARR in H is bounded as:

```
g(n) <= n log n
```

Hence, the running time of the first phase of the heap sort algorithm is O(n log n).

In phase 2, we have H which is a complete tree with m elements having left and right sub-trees as heaps. Assuming L to be the root of the tree, *reheaping* the tree would need 4 comparisons to move L one step down the tree H. Since the depth of H cannot exceed O(log m), reheaping the tree will require a maximum of 4 log m comparisons to find the right location of L in H.

Since n elements will be deleted from heap H, reheaping will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

```
h(n) <= 4n log n
```

Hence, the running time of the second phase of the heap sort algorithm is O(n log n).

Each phase requires time proportional to O(n log n). Therefore, the running time to sort an array of n elements in the worst case is proportional to O(n log n).

Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

PROGRAMMING EXAMPLE

11. Write a program to implement heap sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
```

```
void RestoreHeapUp(int *,int);
void RestoreHeapDown(int*,int,int);
int main()
{
        int Heap[MAX],n,i,j;
        clrscr();
        printf("\n Enter the number of elements : ");
        scanf("%d",&n);
        printf("\n Enter the elements : ");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&Heap[i]);
                RestoreHeapUp(Heap, i);   // Heapify
        }
        // Delete the root element and heapify the heap
        j=n;
        for(i=1;i<=j;i++)
        {
                int temp;
                temp=Heap[1];
                Heap[1]= Heap[n];
                Heap[n]=temp;
                n = n-1;         // The element Heap[n] is supposed to be deleted
                RestoreHeapDown(Heap,1,n); // Heapify
        }
        n=j;
        printf("\n The sorted elements are: ");
        for(i=1;i<=n;i++)
                printf("%4d",Heap[i]);
        return 0;
}
void RestoreHeapUp(int *Heap,int index)
{
        int val = Heap[index];
        while( (index>1) && (Heap[index/2] < val) )// Check parent's value
        {
                Heap[index]=Heap[index/2];
                index /= 2;
        }
        Heap[index]=val;
}
void RestoreHeapDown(int *Heap,int index,int n)
{
        int val = Heap[index];
        int j=index*2;
        while(j<=n)
        {
                if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
                        j++;
                if(Heap[j] < Heap[j/2])   // Check parent's value
                break;
                Heap[j/2]=Heap[j];
                j=j*2;
        }
        Heap[j/2]val;
}
```

14.14 SHELL SORT

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:

- First, insertion sort works well when the input data is 'almost sorted'.
- Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of Shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already 'almost sorted', and hence it provides good performance.

If we take a scenario in which the smallest element is stored in the other end of the array, then sorting such an array with either bubble sort or insertion sort will execute in $O(n^2)$ time and take roughly n comparisons and exchanges to move this value all the way to its correct position. On the other hand, Shell sort first moves small values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

Technique

To visualize the way in which shell sort works, perform the following steps:

- *Step 1:* Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).
- *Step 2:* Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Note that we are only visualizing the elements being arranged in a table, the algorithm does its sorting in-place.

Example 14.8 Sort the elements given below using shell sort.

```
63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33
```

Solution

Arrange the elements of the array in the form of a table and sort the columns.

```
63  19  7   90  81  36  54  45          Result:
72  27  22  9   41  59  33              63  19  7   9   41  36  33  45
                                        72  27  22  90  81  59  54
```

The elements of the array can be given as:

```
63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54
```

Repeat Step 1 with smaller number of long columns.

```
63  19  7   9   41                      Result:
36  33  45  72  27                      22  19  7   9   27
22  90  81  59  54                      36  33  45  59  41
                                        63  90  81  72  54
```

The elements of the array can be given as:

```
22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54
```

Repeat Step 1 with smaller number of long columns.

			Result:		
22	19	7	9	19	7
9	27	36	22	27	36
33	45	59	33	45	54
41	63	90	41	63	59
81	72	54	81	72	90

The elements of the array can be given as:

```
9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90
```

Finally, arrange the elements of the array in a single column and sort the column.

	Result:
9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

Finally, the elements of the array can be given as:

```
7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90
```

The algorithm to sort an array of elements using shell sort is shown in Fig. 14.13. In the algorithm, we sort the elements of the array `Arr` in multiple passes. In each pass, we reduce the `gap_size` (visualize it as the number of columns) by a factor of half as done in Step 4. In each iteration of the `for` loop in Step 5, we compare the values of the array and interchange them if we have a larger value preceding the smaller one.

```
Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3:     SET FLAG = 0
Step 4:     SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5:     Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)
Step 6:           IF Arr[I + GAP_SIZE] > Arr[I]
                        SWAP Arr[I + GAP_SIZE], Arr[I]
                        SET FLAG = 0
Step 7: END
```

Figure 14.13 Algorithm for shell sort

PROGRAMMING EXAMPLE

12. Write a program to implement shell sort algorithm.

```
#include<stdio.h>
void main()
{
        int arr[10]={-1};
        int i, j, n, flag = 1, gap_size, temp;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter %d numbers: ",n); // n was added
        for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
        gap_size = n;
        while(flag == 1 || gap_size > 1)
        {
                flag = 0;
                gap_size = (gap_size + 1) / 2;
                for(i=0; i< (n - gap_size); i++)
                {
                        if( arr[i+gap_size] < arr[i])
                        {
                                temp = arr[i+gap_size];
                                arr[i+gap_size] = arr[i];
                                arr[i] = temp;
                                flag = 0;
                        }
                }
        }
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++){
                printf(" %d\t", arr[i]);
        }
}
```

14.15 TREE SORT

A tree sort is a sorting algorithm that sorts numbers by making use of the properties of binary search tree (discussed in Chapter 10). The algorithm first builds a binary search tree using the numbers to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order. We will not discuss this topic in detail here because we assume that reader has already studied it in sufficient details in the Chapter 10.

Complexity of Tree Sort Algorithm

Let us study the complexity of tree sort algorithm in all three cases.

Best Case	Worst Case
• Inserting a number in a binary search tree takes O (log n) time. • So, the complete binary search tree with n numbers is built in O (n log n) time. • A binary tree is traversed in O (n) time. • Total time required = O (n log n) + O (n) = O (n log n)	• Occurs with an unbalanced binary search tree, i.e., when the numbers are already sorted. • Binary search tree with n numbers is built in O (n^2) time. • A binary tree is traversed in O (n) time. • Total time required = O(n^2) + O(n) = O(n^2). • The worst case can be improved by using a self-balancing binary search tree.

PROGRAMMING EXAMPLE

13. Write a program to implement tree sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct tree
{
        struct tree *left;
        int num;
        struct tree *right;
} ;
void insert (struct tree **, int);
void inorder (struct tree *);
void main( )
{
        struct tree *t ;
        int arr[10];
        int i ;
        clrscr( ) ;
        printf("\n Enter 10 elements : ");
        for(i=0;i<10;i++)
                scanf("%d", &arr[i]);
        t = NULL ;
        printf ("\n The elements of the array are : \n" ) ;
        for (i = 0 ; i <10 ; i++)
                printf ("%d\t", arr[i]) ;
        for (i = 0 ; i <10 ; i++)
                insert (&t, arr[i]) ;
        printf ("\n The sorted array is : \n") ;
        inorder (t ) ;
        getche( ) ;
}
void insert (struct tree **tree_node, int num)
{
        if ( *tree_node == NULL )
        {
                *tree_node = malloc (sizeof ( struct tree )) ;
                ( *tree_node ) -> left = NULL ;
                ( *tree_node ) -> num = num ;
                ( *tree_node ) -> right = NULL ;
        }
        else
        {
                if ( num < ( *tree_node ) -> num )
                        insert ( &( ( *tree_node ) -> left ), num ) ;
                else
                        insert ( &( ( *tree_node ) -> right ), num ) ;
        }
}
void inorder (struct tree *tree_node )
{
        if ( tree_node != NULL )
        {
                inorder ( tree_node -> left ) ;
                printf ( "%d\t", tree_node -> num ) ;
                inorder ( tree_node -> right ) ;
        }
}
```

Table 14.1 Comparison of algorithms

Algorithm	Average Case	Worst Case
Bubble sort	$O(n^2)$	$O(n^2)$
Bucket sort	O(n.k)	$O(n^2.k)$
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$
Shell sort	–	$O(n \log^2 n)$
Merge sort	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)
Quick sort	O(n log n)	$O(n^2)$

14.16 COMPARISON OF SORTING ALGORITHMS

Table 14.1 compares the average-case and worst-case time complexities of different sorting algorithms discussed so far.

14.17 EXTERNAL SORTING

External sorting is a sorting technique that can handle massive amounts of data. It is usually applied when the data being sorted does not fit into the main memory (RAM) and, therefore, a slower memory (usually a magnetic disk or even a magnetic tape) needs to be used. We will explain the concept of external sorting using an example discussed below.

Example 14.9 Let us consider we need to sort 700 MB of data using only 100 MB of RAM. The steps for sorting are given below.

Step 1: Read 100 MB of the data in RAM and sort this data using any conventional sorting algorithm like quick sort.

Step 2: Write the sorted data back to the magnetic disk.

Step 3: Repeat Steps 1 and 2 until all the data (in 100 MB chunks) is sorted. All these seven chunks that are sorted need to be merged into one single output file.

Step 4: Read the first 10 MB of each of the sorted chunks and call them input buffers. So, now we have 70 MB of data in the RAM. Allocate the remaining RAM for output buffer.

Step 5: Perform seven-way merging and store the result in the output buffer. If at any point of time, the output buffer becomes full, then write its contents to the final sorted file. However, if any of the 7 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or else mark the input buffer (sorted chunk) as exhausted if it does not has any more left with it. Make sure that this chunk is not used for further merging of data.

The external merge sorting can be visualized as given in Fig. 14.14.

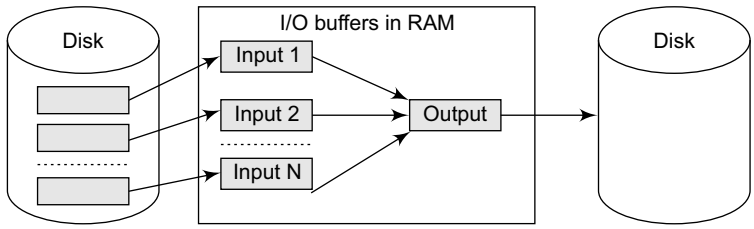


Figure 14.14 External merge sorting

Generalized External Merge Sort Algorithm

From the example above, we can now present a generalized merge sort algorithm for external sorting. If the amount of data to be sorted exceeds the available memory by a factor of K, then K chunks (also known as K run lists) of data are created. These K chunks are sorted and then a K-way merge is performed. If the amount of RAM available is given as X, then there will be K input buffers and 1 output buffer.

In the above example, a single-pass merge was used. But if the ratio of data to be sorted and available RAM is particularly large, a multi-pass sorting is used. We can first merge only the first half of the sorted chunks, then the other half, and finally merge the two sorted chunks. The exact number of passes depends on the following factors:

- Size of the data to be sorted when compared with the available RAM
- Physical characteristics of the magnetic disk such as transfer rate, seek time, etc.

Applications of External Sorting

External sorting is used to update a master file from a transaction file. For example, updating the `EMPLOYEES` file based on new hires, promotions, appraisals, and dismissals.

It is also used in database applications for performing operations like *Projection* and *Join*. Projection means selecting a subset of fields and join means joining two files on a common field to create a new file whose fields are the union of the fields of the two files. External sorting is also used to remove duplicate records.

Points to Remember

- Searching refers to finding the position of a value in a collection of values. Some of the popular searching techniques are linear search, binary search, interpolation search, and jump search.
- Linear search works by comparing the value to be searched with every element of the array one by one is a sequence until a match is found.
- Binary search works efficiently with a sorted list. In this algorithm, the value to be searched is compared with the middle element of the array segment.
- In each step of interpolation search, the search space for the value to be found is calculated. The calculation is done based on the values at the bounds of the search space and the value to be searched.
- Jump search is used with sorted lists. We first check an element and if it is less than the desired value, then a block of elements is skipped by jumping ahead, and the element following this block is checked. If the checked element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element.
- Internal sorting deals with sorting the data stored in the memory, whereas external sorting deals with sorting the data stored in files.
- In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other.
- Insertion sort works by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in the correct place.
- Selection sort works by finding the smallest value and placing it in the first position. It then finds the second smallest value and places it in the second position. This procedure is repeated until the whole array is sorted.
- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. *Divide* means partitioning the `n`-element array to be sorted into two sub-arrays of `n/2` elements in each sub-array. *Conquer* means sorting the two sub-arrays recursively using merge sort. *Combine* means merging the two sorted sub-arrays of size `n/2` each to produce a sorted array of `n` elements. The running time of merge sort in average case and worst case can be given as `O(n log n)`.
- Quick sort works by using a divide-and-conquer strategy. It selects a pivot element and rearranges the elements in such a way that all elements less than pivot appear before it and all elements greater than pivot appear after it.
- Radix sort is a linear sorting algorithm that uses the concept of sorting names in alphabetical order.
- Heap sort sorts an array in two phases. In the first phase, it builds a heap of the given array. In the second phase, the root element is deleted repeatedly and inserted into an array.
- Shell sort is considered as an improvement over insertion sort, as it compares elements separated by a gap of several positions.
- A tree sort is a sorting algorithm that sorts numbers by making use of the properties of binary search tree. The algorithm first builds a binary search tree using the numbers to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order.

EXERCISES

Review Questions

1. Which technique of searching an element in an array would you prefer to use and in which situation?
2. Define sorting. What is the importance of sorting?
3. What are the different types of sorting techniques? Which sorting technique has the least worst case?
4. Explain the difference between bubble sort and quick sort. Which one is more efficient?
5. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using
   (a) insertion sort (b) selection sort
   (c) bubble sort (d) merge sort
   (e) quick sort (f) radix sort
   (g) shell sort
6. Compare heap sort and quick sort.
7. Quick sort shows quadratic behaviour in certain situations. Justify.
8. If the following sequence of numbers is to be sorted using quick sort, then show the iterations of the sorting process.
   42, 34, 75, 23, 21, 18, 90, 67, 78
9. Sort the following sequence of numbers in descending order using heap sort.
   42, 34, 75, 23, 21, 18, 90, 67, 78
10. A certain sorting technique was applied to the following data set,
    45, 1, 27, 36, 54, 90
    After two passes, the rearrangement of the data set is given as below:
    1, 27, 45, 36, 54, 90
    Identify the sorting algorithm that was applied.
11. A certain sorting technique was applied to the following data set,
    81, 72, 63, 45, 27, 36
    After two passes, the rearrangement of the data set is given as below:
    27, 36, 80, 72, 63, 45
    Identify the sorting algorithm that was applied.
12. A certain sorting technique was applied to the following data set,
    45, 1, 63, 36, 54, 90
    After two passes, the rearrangement of the data set is given as below:
    1, 45, 63, 36, 54, 90
    Identify the sorting algorithm that was applied.
13. Write a recursive function to perform selection sort.
14. Compare the running time complexity of different sorting algorithms.
15. Discuss the advantages of insertion sort.

Programming Exercises

1. Write a program to implement bubble sort. Given the numbers 7, 1, 4, 12, 67, 33, and 45. How many swaps will be performed to sort these numbers using the bubble sort.
2. Write a program to implement a sort technique that works by repeatedly stepping through the list to be sorted.
3. Write a program to implement a sort technique in which the sorted array is built one entry at a time.
4. Write a program to implement an in-place comparison sort.
5. Write a program to implement a sort technique that works on the principle of divide and conquer strategy.
6. Write a program to implement partition-exchange sort.
7. Write a program to implement a sort technique which sorts the numbers based on individual digits.
8. Write a program to sort an array of integers in descending order using the following sorting techniques:
   (a) insertion sort (b) selection sort
   (c) bubble sort (d) merge sort
   (e) quick sort (f) radix sort
   (g) shell sort
9. Write a program to sort an array of floating point numbers in descending order using the following sorting techniques:
   (a) insertion sort (b) selection sort
   (c) bubble sort (d) merge sort
   (e) quick sort (f) radix sort
   (g) shell sort
10. Write a program to sort an array of names using the bucket sort.

Multiple-choice Questions

1. The worst case complexity is _____ when compared with the average case complexity of a binary search algorithm.
   (a) Equal (b) Greater
   (c) Less (d) None of these
2. The complexity of binary search algorithm is
   (a) O(n) (b) $O(n^2)$
   (c) O(n log n) (d) O(log n)
3. Which of the following cases occurs when searching an array using linear search the value to be searched is equal to the first element of the array?
   (a) Worst case (b) Average case
   (c) Best case (d) Amortized case
4. A card game player arranges his cards and picks them one by one. With which sorting technique can you compare this example?
   (a) Bubble sort (b) Selection sort
   (c) Merge sort (d) Insertion sort
5. Which of the following techniques deals with sorting the data stored in the computer's memory?
   (a) Insertion sort (b) Internal sort
   (c) External sort (d) Radix sort
6. In which sorting, consecutive adjacent pairs of elements in the array are compared with each other?
   (a) Bubble sort (b) Selection sort
   (c) Merge sort (d) Radix sort
7. Which term means sorting the two sub-arrays recursively using merge sort?
   (a) Divide (b) Conquer
   (c) Combine (d) All of these
8. Which sorting algorithm sorts by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place?
   (a) Insertion sort (b) Internal sort
   (c) External sort (d) Radix sort
9. Which algorithm uses the divide, conquer, and combine algorithmic paradigm?
   (a) Selection sort (b) Insertion sort
   (c) Merge sort (d) Radix sort
10. Quick sort is faster than
    (a) Selection sort (b) Insertion sort
    (c) Bubble sort (d) All of these
11. Which sorting algorithm is also known as tournament sort?
    (a) Selection sort (b) Insertion sort
    (c) Bubble sort (d) Heap sort

True or False

1. Binary search is also called sequential search.
2. Linear search is performed on a sorted array.
3. For insertion sort, the best case occurs when the array is already sorted.
4. Selection sort has a linear running time complexity.
5. The running time of merge sort in the average case and the worst case is `O(n log n)`.
6. The worst case running time complexity of quick sort is `O(n log n)`.
7. Heap sort is an efficient and a stable sorting algorithm.
8. External sorting deals with sorting the data stored in the computer's memory.
9. Insertion sort is less efficient than quick sort, heap sort, and merge sort.
10. The average case of insertion sort has a quadratic running time.
11. The partitioning of the array in quick sort is done in `O(n)` time.

Fill in the Blanks

1. Performance of the linear search algorithm can be improved by using a _____.
2. The complexity of linear search algorithm is _____.
3. Sorting means _____.
4. _____ sort shows the best average-case behaviour.
5. _____ deals with sorting the data stored in files.
6. $O(n^2)$ is the running time complexity of _____ algorithm.
7. In the worst case, insertion sort has a _____ running time.
8. _____ sort uses the divide, conquer, and combine algorithmic paradigm.
9. In the average case, quick sort has a running time complexity of _____.
10. The execution time of bucket sort in average case is _____.
11. The running time of merge sort in the average and the worst case is _____.
12. The efficiency of quick sort depends on _____.